A Robot Path Planning Framework that Learns from Experience

Dmitry Berenson Pieter Abbeel Ken Goldberg

University of California, Berkeley, Berkeley, CA, USA
{berenson, pabbeel}@eecs.berkeley.edu, goldberg@berkeley.edu

***Abstract*— We propose a framework, called *Lightning*, for planning paths in high-dimensional spaces that is able to learn from experience, with the aim of reducing computation time. This framework is intended for manipulation tasks that arise in applications ranging from domestic assistance to robot-assisted surgery. Our framework consists of two main modules, which run in parallel: a planning-from-scratch module, and a module that retrieves and repairs paths stored in a path library. After a path is generated for a new query, a library manager decides whether to store the path based on computation time and the generated path's similarity to the retrieved path. To retrieve an appropriate path from the library we use two heuristics that exploit two key aspects of the problem: (i) A correlation between the amount a path violates constraints and the amount of time needed to repair that path, and (ii) the implicit division of constraints into those that vary across environments in which the robot operates and those that do not.**

We evaluated an implementation of the framework on several tasks for the PR2 mobile manipulator and a minimally-invasive surgery robot in simulation. We found that the retrieve-and-repair module produced paths faster than planning-from-scratch in over 90% of test cases for the PR2 and in 58% of test cases for the minimally-invasive surgery robot.

I. Introduction

A long-standing goal of research in robotics is to create a robot whose ability to perform a task improves with experience through performing similar tasks. For example, consider a mobile manipulator operating in a domestic environment. Such a robot may frequently be expected to retrieve an item from a cupboard. The most common approaches to solving such tasks is to construct a motion plan using no prior knowledge of the task and the environment; a planning algorithm is only given the robot and environment models along with the start and goal configurations and asked to generate a path. While this approach, which we term *planning-from-scratch* (PFS), is general, it can produce unacceptably long planning times for difficult problems. Even if the robot previously generated a path for a very similar task, PFS affords no way to take advantage of this previous computation.

We seek to create a framework for planning paths in high-dimensional spaces that is able to improve with experience, with the aim of reducing computation time. This framework is intended to plan paths for manipulators, and it can be applied to problems ranging from mobile manipulation to robot-assisted surgery. Our framework, which we call *Lightning* for its ability to plan quickly, leverages the generality of PFS to produce solutions in new situations and the efficiency of re-using previous experience in situations similar to previously-encountered ones. We integrate these two approaches to create a framework that is both general and efficient. It is also straightforward to enhance the framework with parallelization and cloud computing by parallelizing the retrieval and repair algorithms and by storing the experience of multiple robots in the cloud.

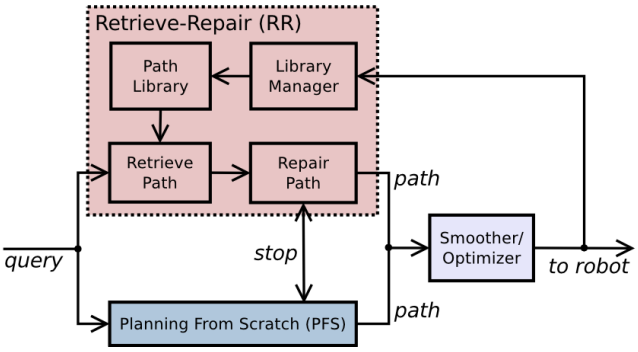

Fig. 1. Diagram of the Lightning framework.

The Lightning framework (see Figure 1) consists of two main modules, which are run in parallel: PFS, and a module that retrieves and repairs paths stored in a path library, which we call Retrieve-Repair (RR). Given a new query, both modules are started simultaneously and the first path produced by either module is executed on the robot while the other module is stopped. After a path is generated, a library manager decides whether to store the path based on the computation times of the two modules and the generated path's similarity to the retrieved path.

The key part of the framework is the RR module, which retrieves a path that is likely to be appropriate for a given scene from the path library and repairs it. To retrieve an appropriate path we exploit two key features of the problem: (i) A correlation between the amount a path violates constraints and the amount of time needed to repair that path, and (ii) the implicit division of constraints into *variant* and *invariant* constraints. These features allow us to formulate two heuristics to retrieve a path from the library: the first quickly selects n candidate paths from the library by measuring the distance between the endpoints of the path and the query start and goal, and the second retrieves the path from the n candidates which has the least constraint violation. Once the RR module retrieves an appropriate path, the path can be repaired using a variety of methods. To guarantee feasibility, we use a path repair method based on bidirectional RRTs.

The Lightning framework has several advantages. First, we do not need to arbitrarily pre-generate some number of

paths before running the system. We can start with an empty path library and build that library as the robot performs its intended tasks. This is a significant advantage because it may be difficult to envision the types of tasks and environments the robot encounters before it actually encounters them. The ability to learn without prior information about the environment is key for a domestic robot because we may not know what a user's kitchen or living room looks like and what objects will be present there a priori.

Also, comparing to a standard sampling-based motion planner, there can be a significant benefit in terms of computation time for difficult problems. Consider the reaching-into-the-cluttered-cupboard example described above. Solving such problems with a sampling-based planner tends to be time-consuming because of the narrowness of the space. However, since the geometry of the kitchen cupboard does not change between queries, previously-computed paths will be very close to valid (we only need to avoid the changing clutter in the cupboard, not navigate into the narrow space of the cupboard). These paths will require little repair and feasible paths will be produced much more quickly than using a sampling-based planner to plan from scratch.

Finally, a major advantage of this system is that we can run the system over the lifetime of the robot, even as the robot transitions between different types of tasks. It may take many queries for the RR module to become proficient at a new type of task, but, in the worst case, we will always perform as well as PFS, which is a state-of-the-art planner.

The contributions of this paper are the following:

1) A general and efficient framework for building and using a path library.
2) A formal definition of the problem of retrieving and repairing a path from the library.
3) A fast method for retrieving paths based on the above definition.
4) Experiments showing a range of situations in which the Lightning framework outperforms PFS.

In the remainder of this paper we describe related work, formally define the path retrieval problem, present a fast method for retrieving and repairing paths from a path library, and give a detailed description of each module in the Lightning framework. We then evaluate an implementation of the framework on tasks for the PR2 mobile manipulator and a minimally-invasive surgery robot. We conclude with a discussion of possible extensions of our framework.

II. Related Work

Reusing previous computation to aid in solving a new query has been studied using several approaches in motion planning and control. A major area of research focuses on how to re-use computation in the presence of moving obstacles to achieve fast replanning [1], [2], [3]. In these scenarios, changes in the environment between successive calls to the planner are assumed to be minor and the goal is fixed, whereas environments in our scenarios can vary dramatically and the start and goal vary as well.

In sampling-based planning, one approach biases sampling during a planning query based on previous experience [4]. However, this approach requires identifying features that capture the quality of a sample, which may be difficult to do in a given problem domain.

Trajectory libraries have been studied extensively as a method for constructing optimal control policies [5], [6], [7], [8]. However, these methods either create a policy that drives the robot toward a fixed goal or they do not account for varying environment geometry. Stolle et al. [9] investigated transferring a set of trajectories between different goals and environments, but their method was highly-specialized for planning footstep locations for a quadruped robot.

Using libraries of paths has been previously investigated as an approach to local planning [10], [11], [12]. These approaches iteratively select a short path from the library that brings the robot closer to a goal or to a predefined path. Similar approaches have also been developed in the graphics literature [13], [14], where short recordings of motion-capture data are stitched together to produce a complex sequence of motions for an animated character. A related approach uses a library of motion primitives to generate transitions between humanoid hand and foot placements on rough terrain [15]. Our approach differs in that we are concerned with selecting a single path from the library to bring the robot from the start to the goal.

Similar to our approach, Jetchev and Toussaint [16] use a library of paths to aid in computing a new path in a new environment. However, their paths are computed in end-effector space, whereas ours reside in the higher-dimensional C-space. Their approach also requires warping paths to have common start and goal poses, and it is unclear how this transformation affects the information stored by the paths. Their path library is constructed beforehand and it is static, which assumes that it is possible to envision the range of scenarios the robot encounters a priori, whereas our approach builds the library based on the robot's experience. On the other hand, their method for selecting paths based on environment features could be incorporated into our framework.

Recently Dragan et al. [17] presented a method which selects a goal from a set of possible goals that is likely to yield a good path based on previous experience. While this appears to be a promising approach, it has not yet been extended to consider obstacles en route to the goal configuration, which play a very prominent role in determining the path. However, selecting goals based on previous experience could also be incorporated in our method for retrieving paths from the library.

III. Retrieving and Repairing Paths

To motivate our approach to using a library of paths, let us consider the set of all continuous non-self-intersecting configuration space paths P.[1] Subsets of P discussed in this section are shown in Figure 2.

[1]Note that P is infinite-dimensional

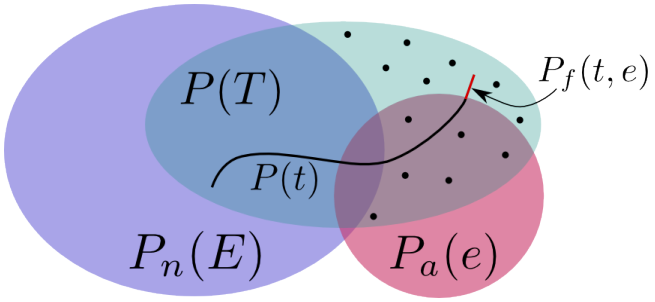

Fig. 2. Illustration of sets in path space for a given task t and environment e. $P(t)$ is the set of paths that accomplish the task t. $P(T)$ is the union of $P(t)$ for all $t \in T$. $P_n(E)$ and $P_a(e)$ are the sets of paths that violate the invariant and variant constraints, respectively. $P_f(t,e)$ is the set of *valid* paths for t. The black points represent paths in the path library $L(T)$.

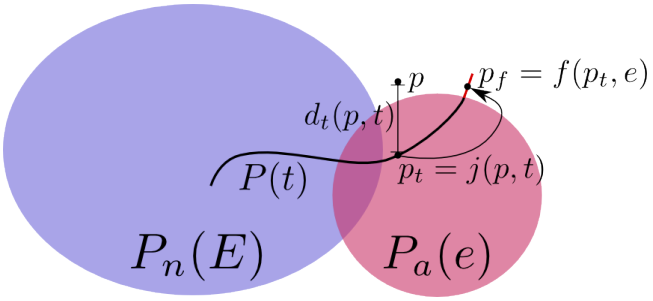

Fig. 3. Illustration of the distance, projection, and repair functions d_t, j, and f, respectively. A path p is projected to $P(t)$ using j and then repaired using f. This process produces a path p_f that accomplishes the task without violating constraints.

A. Definitions

We will refer to a path planning query consisting of a start configuration and at least one goal configuration as t. Let T represent the set of all t the robot is expected to solve. Let $P(t) \subseteq P$ represent all paths that satisfy a given t without regard to any constraints and let $P(T) \subseteq P$ represent the union of all such $P(t)$ for all $t \in T$. We define the set of environments the robot is expected to operate in as E.

Constraints imposed by the environment, task, robot geometry, and kinematics may invalidate paths in P. For example a path $p \in P(t)$ may be invalid for an environment $e \in E$ because of collision with the environment or may be invalid for all environments because of robot self-collision. We will divide the constraints into two types: those which are present for all $e \in E$, which we call *invariant* constraints, and those which are present for a given e but not all $e \in E$, which we call *variant* constraints. Examples of invariant constraints are robot self-collision, joint limits, and areas in the environment that are occupied in all E. Examples of variant constraints are areas of the environment that are sometimes, but not always, occupied. Note that environments are considered in the robot's frame, so environments with similar geometry relative to the robot will produce similar constraints, independent of the robot's location in the world.

Our approach does not require that we specify the variant and invariant constraints explicitly. Rather, these constraint types emerge implicitly when building a path library. We will use this fact to create a heuristic for selecting paths from the path library in Section III-B.

Let the set of paths which violate the invariant constraints be $P_n(E) \subseteq P$ and let the set of paths that violate the variant constraints for a given e be $P_a(e) \subseteq P$.

A given t may induce a $P(t)$ that intersects one or both of $P_n(E)$ and $P_a(e)$. Let us define the set of valid paths for a given task t in a given environment e as

$$P_f(t,e) = \{p \in P(t) \mid p \notin P_n(E), p \notin P_a(e)\}. \quad (1)$$

The standard path planning problem is to find any path $p \in P_f(t,e)$. The most common approach to this problem is to use a planner that iteratively builds the path p from only the start and goal configurations. Depending on the task and environment, building such a path can be computationally expensive. In this paper we evaluate a different approach to the path planning problem, which uses a library of previously-computed paths, with the aim of reducing computation time.

Let us assume that we have a library of paths for a given set of tasks T. We will represent this library as the set $L(T) \subseteq P$. Let us further assume that $L(T) \cap P_n(E) = \emptyset$, i.e., no paths in $L(T)$ violate the invariant constraints (the Lightning framework implicitly constructs such a library).

We will now define functions which allow us to retrieve a path from the library and repair it. Let v be a function which quantifies the degree of violation of the constraints:

$$v : P \times E \to \mathbb{R}_{\geq 0}. \quad (2)$$

Let $v_n(p,E)$ be the violation of the invariant constraints and let $v_a(p,e)$ be the violation of the variant constraints. Further, let $v(p,e) \geq \min(v_n(p,E), v_a(p,e))$. We will use v to help retrieve a path from the library in Section III-B.

Let f be the *repair function*, which has the form

$$f : P(t) \times E \to P_f(t,e). \quad (3)$$

Let this function return $\emptyset$ if it is not able to repair the path. We will discuss the implementation of f in Section IV.

In general, a library $L(T)$ may not contain a $p \in P(t)$ for a given t, i.e., we may not have an example path that accomplishes the task, regardless of the constraints. This implies that we cannot apply the repair function f, whose domain is restricted to $P(t)$. In this case, we must first generate a path in $P(t)$ before applying f. One way to generate such a path is to project a $p \in L(T)$ onto $P(t)$ (see Figure 3). To do this, we first define a function which evaluates the distance between a path p and the set $P(t)$.

Let us denote the start and goal of p as $p(0)$ and $p(1)$, respectively. By definition, $P(t)$ contains all paths that go between the start and goal of t (denoted t_s and t_g, respectively). Thus $P(t)$ must contain the path

$$p_t = \{l(t_s, p(0)), p, l(p(1), t_g)\}, \quad (4)$$

where l is a line segment between the two input configurations. The sum of the lengths of the line segments that need to be added to p to achieve the task can be used as a distance function between p and t:

$$d_t(p,t) = \| t_s - p(0) \| + \| t_g - p(1) \|. \quad (5)$$

$d_t(p,t)$ is 0 if $p \in P(t)$ and increases as the endpoints move farther from those required by t. We can then define a path-space projection function

$$j : P \times T \to P(t), \quad (6)$$

Algorithm 1: NaiveRetrieveRepair(t,e,$L(T)$)

for all $p \in L(T)$ **do**
  $p_t \leftarrow j(p,t)$;
  $p_f \leftarrow f(p_t,e)$;
  if $p_f \neq \emptyset$ **then**
    return p_f;
  end
end
return *Failure*;

which produces the $p_t \in P(t)$ in Equation 4 from any input path p and t (see Figure 3).

Given the above functions, a naive algorithm (Alg. 1) to generate a $p \in P_f(t,e)$ using a $p \in L(T)$ is readily apparent.

B. Considering Computation Time

Algorithm 1 is straightforward but highly impractical when we consider the computation time necessary to perform each step. Let the computation time necessary to perform $j(p,t)$, $f(p,e)$, and $v(p,e)$ be $\tau_j(p,t)$, $\tau_f(p,e)$, and $\tau_v(p,e)$, respectively. Since computing $j(p,t)$ only requires appending two line segments to p, $\tau_j(p,t) \approx 0$. $\tau_f(p,e)$ is difficult to characterize, since it depends strongly on how it is implemented. f can be a global path planner that searches the entire configuration space, or it can deform a path locally. The global planner will likely take more time but return $\emptyset$ less often, while the local planner will likely take less time but return $\emptyset$ more often. Regardless of which approach is used, we assume that repairing a path is much more time-consuming than evaluating the violation of a path (i.e. $\tau_f(p,e) \gg \tau_v(p,e)$), which is true for both of the above approaches on practical path planning problems.

Thus to reduce computation time, we would like to minimize $\tau_f(p,e)$. Our approach will be to use v as a heuristic for $\tau_f(p,e)$. This heuristic assumes that for $p_1, p_2 \in P(t)$

$$v(p_1,e) \geq v(p_2,e) \Rightarrow \tau_f(p_1,e) \geq \tau_f(p_2,e). \quad (7)$$

The motivation behind this heuristic is that paths which violate fewer constraints take less time to repair.

If Equation 7 holds, we can produce a $p_f \in P_f(t,e)$ in minimal time using only one evaluation of f. We do this by computing $p_f = f(p_t^*, e)$, where

$$p_t^* = \underset{p_t}{\operatorname{argmin}} \; v(p_t,e). \quad (8)$$

Thus, taking into account the high computation time necessary to compute f, we can propose another algorithm (Alg. 2) to find a path in $P_f(t,e)$ using a path from $L(T)$.

While effective, this algorithm requires evaluating $v(p_t,e)$ for every path in $L(T)$. While we can reasonably assume that $\tau_f(p,e) \gg \tau_v(p,e)$, it would not be appropriate to assume that $\tau_v(p,e)$ is negligible for practical path planning problems. Thus this algorithm may also be impractical depending on the number of paths in $L(T)$ and the computational resources available. Let us assume that our computational resources allow us to perform n evaluations of v in parallel. To minimize computation time, we would like to select the

Algorithm 2: NaiveRetrieveRepair2(t,e,$L(T)$)

$P_t \leftarrow \emptyset$;
for all $p \in L(T)$ **do**
  $P_t \leftarrow P_t \cup j(p,t)$;
end
$p_t^* \leftarrow \underset{p_t \in P_t}{\operatorname{argmin}} \; v(p_t,e)$;
$p_f = f(p_t^*, e)$;
if $p_f \neq \emptyset$ **then**
  return p_f;
return *Failure*;

n paths for evaluation which are most likely to yield fast evaluations of f. However, without evaluating v, we do not know which path will be likely to yield a low $\tau_f(p,e)$.

We can create another heuristic to decide which paths will likely have less violation. This heuristic is based on the violation of the invariant constraints $v_n(p,E)$ as a result of the projection: A $p \in L(T)$ which yields a smaller $d_t(p,t)$ will likely produce a smaller $v_n(p,E)$ when projected to t. I.e. for $p_1, p_2 \in L(T)$

$$d_t(p_1,t) \geq d_t(p_2,t) \Rightarrow v_n(j(p_1,t),e) \geq v_n(j(p_2,t),e). \quad (9)$$

This heuristic is most appropriate when the constraints are evaluated as a function of configuration. In such cases, the projection, which only adds configurations to a path, can only increase the violation of constraints; the more configurations added, the more potential for violation of the constraints. Since all paths in $L(T)$ do not violate the invariant constraints, the shorter the projection, the less likely the resulting path is to violate the invariant constraints. Since this heuristic does not consider violation of the variant constraints (there are no guarantees on variant constraint violation for paths in $L(T)$), we will only use it to narrow the number of candidates for violation evaluation, not to select a path directly.

Given the above heuristic, we arrive at Algorithm 3, which only evaluates v on n paths chosen from $L(T)$.

This algorithm selects n paths to project to t based on their distance to t. It then evaluates the violation of these n paths and applies the function f to the path with minimal violation. The f function then produces a path $p_f \in P_f(t,e)$, which solves the task t without violating constraints.

This algorithm is built on two heuristics, which may not hold for a given set of tasks. For instance, even though a path has little violation, the violation may occur in a very cluttered area of the environment and it will require more time to repair this path than one which has more violation in an open area of the environment. Likewise, a projection to $P(t)$ may be short but induce more collisions than a longer projection which happens to avoid obstacles. However, we show that the algorithm built on these heuristics is effective on several real-world planning problems, suggesting that the heuristics do indeed capture the structure of some practical path planning problems.

It is straightforward to extend this algorithm to consider tasks that afford multiple goals, which are common in

Algorithm 3: RetrieveRepair(t,e,n,$L(T)$)

Retrieve:
  for all $p_i \in L(T)$ **do**
    $d_i \leftarrow d_t(p_i, t)$
  end
  $P_n \leftarrow$ Set of n paths from $L(T)$ with n smallest d_i;
  $P_t \leftarrow \emptyset$;
  for all $p \in P_n$ **do**
    $P_t \leftarrow P_t \cup j(p, t)$
  end
  $p_t^* \leftarrow \underset{p_t \in P_t}{\operatorname{argmin}} \; v(p_t, e)$;
Repair:
  $p_f = f(p_t^*, e)$;
  if $p_f \neq \emptyset$ **then**
    return p_f;
  return *Failure*;

planning for manipulators. To consider multiple goals, we generalize $d_t(p, t)$ to compute the distance between a path and all goals allowed by t paired with the start configuration. $d_t(p, t)$ then returns the lowest distance for any pair. $j(p, t)$ is generalized in the same way.

In the following section we provide a description of our implementation of the modules of the Lightning framework. We emphasize that this is only one embodiment of the framework and that there are many ways to implement these modules. We chose these implementations because they performed well in terms of computation time.

IV. The Lightning Framework

An overview of the Lightning framework is shown in Figure 1. Given a path planning query, we run two modules in parallel: planning-from-scratch (PFS) and retrieve-repair (RR). The PFS module attempts to use a planner to find a path from start to goal, while the RR module attempts to find a path by retrieving an appropriate path from a path library and repairing that path so that it does not violate any constraints. PFS and RR run simultaneously, and the first module to finish sends a *stop* signal to the other, which terminates the other module. The path produced by the first module to finish is then sent to a smoother/optimizer. After smoothing/optimization, the path is executed and sent to the library manager within the RR module. We describe the implementation of each module below.

A. Library Manager

The library manager decides whether to insert a path into the path library. It uses the following rules to make this decision: If PFS was the first to produce a path, this means that RR did not have a path sufficiently appropriate for the given query (the ideal path would require no repair, and thus the *repair path* sub-module would not require any time). Thus, to improve the RR module's performance on similar queries, we add the path produced by PFS to RR's path library. If RR produces a path before PFS, then RR has already surpassed PFS's ability on the given query, so there is no need to add RR's path into the library if we are only interested in surpassing PFS. However, there may be some situations where PFS's performance is poor, and simply surpassing PFS is not sufficient in terms of computation time. Thus, if RR was the first to produce a path and that path was significantly different[2] from the path it chose to re-use, we add that path to RR's path library. In this way, RR is able to surpass PFS's performance on difficult problems while not cluttering the path library with redundant paths.

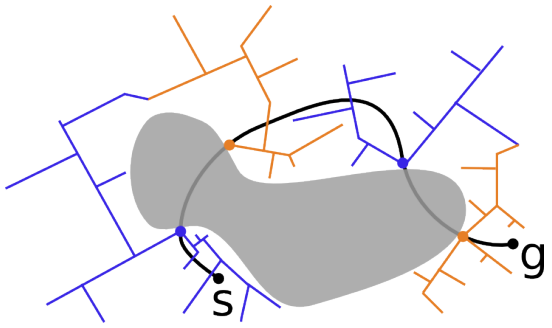

Fig. 4. Repairing an infeasible path using BiRRTs. Blue: Forward-searching trees. Orange: Backward-searching trees.

B. Planning from Scratch

We implement the planning-from-scratch module as a BiDirectional RRT (BiRRT) [19]. We selected RRTs for their ability to explore C-space while retaining an element of "greediness" in the search for a solution. The greedy element is most evident in this bidirectional version of the RRT algorithm, where two trees, one grown from the start and one grown from the goal, take turns exploring the space and attempting to connect to each other.

C. Retrieving and Repairing a Path

To retrieve a path from the library, we perform the "Retrieve" section of Algorithm 3. The v function, which computes constraint violation of a path, is implemented by checking environment collision, joint-limits, and self-collision for the discretized path. v returns the number of configurations which violate any of these constraints.

After retrieving the path with minimum v (from the n paths we evaluate), we apply an implementation of the f function described above to repair the retrieved path, thus generating a path that does not violate constraints. There are several methods in motion planning [3] and trajectory optimization [20] [21] that are capable of this task. Since our aim is to reduce computation time, we use what we believe to be the fastest approach: repairing the path using multiple BiRRTs. This method is also guaranteed to find a solution if one exists. The implementation of this method is as follows:

First we check each point in the discretized path for constraint violation. Any segments of the path that do not violate constraints are preserved. To get from one valid segment to another, we run a BiRRT, whose start tree is rooted at the end of one valid segment and whose goal tree

[2]We evaluate path similarity by treating the paths as time-series and computing the similarity score using Dynamic Time Warping [18].

is rooted at the beginning of the next valid segment (see Figure 4). The composite path consisting of originally-valid path segments and path segments produced by the BiRRT is then returned as the path.

D. Smoother/Optimizer

The smoother/optimizer is implemented using path short-cutting. We make one improvement to the standard path-short-cutting algorithm: We don't attempt a short-cut if the segment of the path between the selected nodes is already straight or nearly straight. Implementing this simple check allowed us to save a great deal of time in smoothing.

V. Results

We are interested in applying the Lightning framework to problems that involve path planning for manipulators. To that end, we evaluate the framework's performance in two domains: reaching tasks for the PR2 in a kitchen scenario and handover tasks for a minimally-invasive surgery robot in a heart-surgery scenario, both in simulation. Since PFS is currently the most common and accepted method for path planning, the key question we wish to answer is: Does using the RR module outperform PFS as more paths are added to the library? And if so, by how much? Thus, in both scenarios, we compare the performance of the PFS and RR modules (for varying library sizes) in terms of computation time.

We used $n = 10$, a BiRRT step size of 0.05rad, and a dynamic time warping threshold of 5 in all experiments. All tests were performed using a computer with 4 3.4GHz cores and 16GB of RAM. We describe our test scenarios below.

A. PR2 in a Kitchen Scenario

The Willow Garage PR2 robot is a two-arm mobile manipulator intended for use in domestic environments. In our scenario, the PR2 is placed in a kitchen, where its task is to reach and grasp a bottle in cluttered scenes using the 7DOF right arm. The poses of the bottle, several box obstacles, and the PR2's base are randomly determined for each problem instance. The bottles and boxes were placed randomly on the counter/cupboard shelf, while the PR2's base was placed randomly within several centimeters of a nominal pose in front of the counter/cupboard. C-space goals are generated by sampling over grasping poses around the top of the bottle and performing inverse kinematics (IK). The collision-free IK solutions are passed to the RR module as goals.

We consider two types of scenes in the kitchen: grasping a bottle on a counter with three box obstacles and grasping a bottle in a cupboard with two box obstacles (Figure 5), which we consider to be a more difficult planning problem because of the narrowness of the cupboard.

B. Minimally-invasive Heart Surgery Scenario

We also consider a minimally-invasive surgical procedure where two small manipulators and a camera are tele-operated by a surgeon to perform an operation on the heart. A common procedure of this type is Coronary Artery Bypass Graft (CABG) surgery. In this procedure, the left lung is deflated,

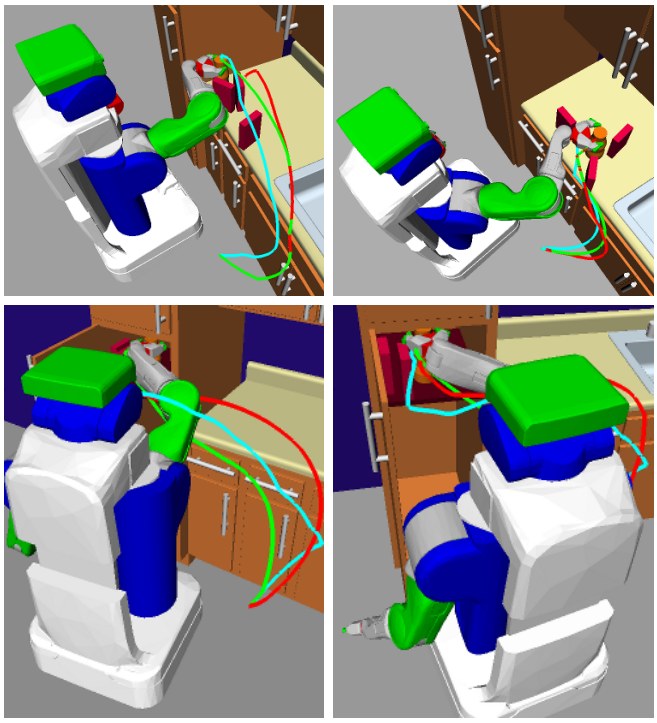

Fig. 5. Examples where PR2 is reaching for a bottle on a kitchen counter (above) and in a cupboard (below). Smoothed paths generated by the RR and PFS modules for two tasks are displayed above and two views of the same task are displayed below (only the path of the end-effector is shown). Red: Path chosen from the path library. Green: Path produced by the RR. Light Blue: Path produced by the PFS.

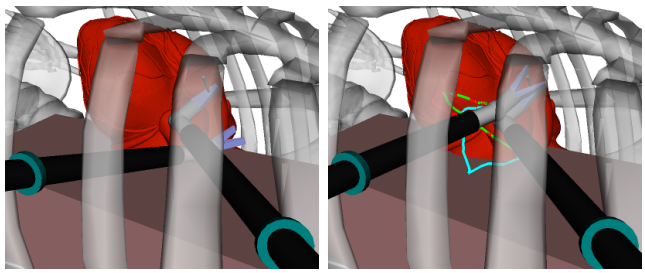

Fig. 6. Example of the minimally-invasive surgery needle-handover task. The left lung is deflated before the procedure. Left: Starting configuration. Right: Paths and end configuration. Red: Path chosen from path library. Green: Path produced by the RR. Light Blue: Path produced by the PFS.

the manipulators are inserted between the ribs into the chest cavity, and the surgeon's task is to graft an artery onto the heart. We seek to automate some of the sub-tasks of this surgery, and here we concentrate on the sub-task of handing-over a suturing needle between manipulators (see Figure 6).

We consider the following instance of the hand-over task: The right manipulator is holding a suturing needle. Both manipulators are placed in a random collision-free configurations near the heart. The task is then to construct a path for the left manipulator that moves it to a configuration where it can grasp the needle (this configuration is generated using IK). The path must avoid collisions with the complex geometry of the heart and ribs while also avoiding the other manipulator. Each manipulator is rooted at its entry port and has 7DOF: 3 rotational joints at the entry port, 1 prismatic joint that allows sliding in and out of the entry port, and the

3 rotational joints at the wrist.

C. Test Results

To evaluate the performance of our framework on the above scenarios, we first populate a path library for each scenario using the Lightning framework. To do this, we run Lightning over thousands of randomly-generated instances of each scenario. We then compare RR and PFS in terms of planning time on each scenario for varying library sizes.

We found that, although instances of each scenario were visually similar, the planning times for the different instances varied widely. This is because it is possible to create situations where the manipulator is fairly unrestricted at the goal and instances where the manipulator must fit into a very tight space to reach the goal in each scenario (see Figure 5). The randomized nature of the path repair and PFS modules also contributed to this high variance. Thus, we find it informative to present the statistics in terms of which module (RR or PFS) performed best in a given instance. Figures 9 and 7 display the percent of instances where RR outperformed PFS for different library sizes.

It may be surprising that, even with a small library, RR was able to achieve a large improvement over PFS (see Figure 7). The reason for this result is that even paths that are far from accomplishing the task still avoid invariant constraints. For instance, paths for the cupboard scenario will bring the end-effector into the cupboard while either colliding slightly with the cupboard (because the base position varies when generating the paths) or avoiding it entirely. Thus, while PFS must find a way to bring the end-effector from the robot's side into the narrow space of the cupboard, RR only needs to repair a fraction of the path where the end-effector is already inside the cupboard. As RR gains more experience, the amount of path needed to repair usually decreases, and for the larger library sizes, RR outperforms PFS in over 90% of instances for both PR2 scenarios.

Figure 8 shows histograms of PFS runtime minus RR runtime for 200 problem instances corresponding to several library sizes for the PR2 scenarios. We used a timeout value of 60 seconds for both modules. 60s was used as the value for a test which timed out when computing the histograms.

As the histograms show, the RR module outperformed PFS by 6.8s and 16.4s on average when using the largest library sizes for the counter and cupboard scenarios, respectively. There were also a significant number of cases where RR outperformed PFS by a very large amount: by over 10s in 18% of cases, and by over 20s in 25% of cases using the largest libraries for the counter and cupboard scenarios, respectively. Average planning times for PFS were 6.80s, 12.2s, and 0.220s for the counter, cupboard, and surgery scenarios, respectively. Average planning times for RR were 1.46s, 0.93s, and 0.084s (using the largest library sizes).

The performance difference was not as prominent for the minimally-invasive surgery scenario, although RR did significantly outperform PFS (see Figure 9). The reason for this small performance difference is that many of the randomly-generated instances were quite easy and required very little

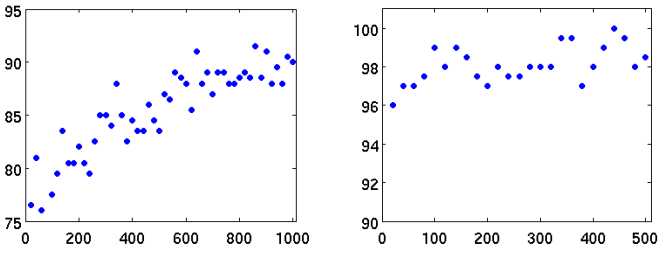

Fig. 7. Percent of instances for the two PR2 scenarios where RR outperformed PFS (y-axis) for varying path library size (x-axis). Left: Kitchen counter scenario. Right: Kitchen cupboard scenario.

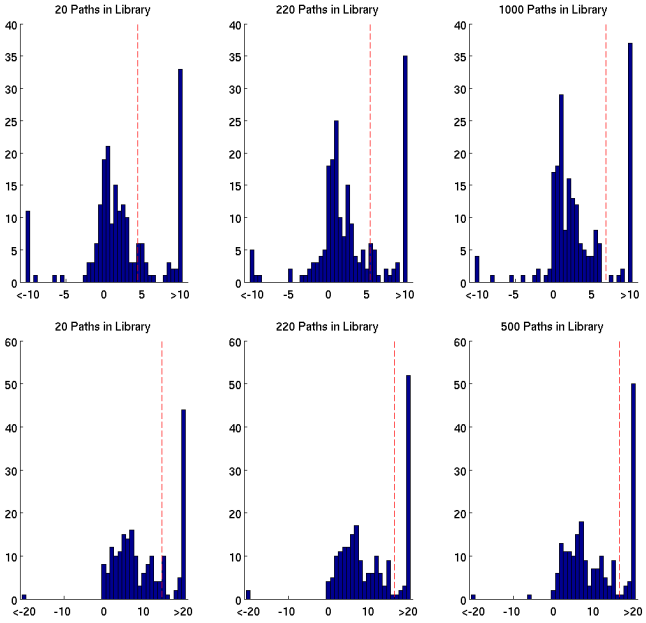

Fig. 8. Histograms of PFS runtime minus RR runtime for PR2 in the kitchen (top row: counter, bottom row: cupboard). The y-axis shows the number of instances (out of 200) that had the computation time difference (in seconds) shown on the x-axis. The red dashed line is the mean.

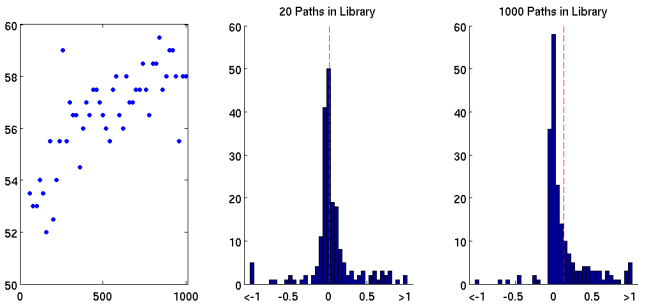

Fig. 9. Left: Percent of instances for needle-handover scenario where RR outperformed PFS for varying path library size. Center and Right: Histograms of PFS runtime minus RR runtime (in seconds) for 200 instances of this scenario.

planning to solve. In these cases, the time necessary to check d_t for all paths in the library and compute v for n paths exceeded the runtime of the PFS module (even though path repair was often not required because a path with no violation was retrieved). If we eliminate the time it took to retrieve a path (which totaled to about 0.046s on average) then the RR module outperformed PFS in 94% of instances for a library

size of 1000 paths for the surgical handover scenario.

These results confirm our hypothesis that the RR module improves with experience and outperforms using PFS alone. In some cases the improvement over PFS is more pronounced than in others, however RR was able to outperform PFS in the majority of instances for each scenario.

VI. Discussion

A. Multiple PFS Modules

It is straightforward to extend this framework to include multiple PFS modules, as long we have the computational resources to run them all in parallel. Having multiple PFS modules would allow us to take advantage of the strengths of many different planning algorithms. We could also vary parameter values over multiple instances of the same algorithm. To accommodate these modules, we can easily generalize the rule for adding a new path to the library: If any PFS module generates a path before RR, all modules are stopped and the path is added to the path library.

B. Parallel/Cloud Computing

Our framework can also benefit greatly from more parallelization and cloud computing. One application would be storing the path library in the cloud and allowing multiple robots of the same type operating in similar environments (for example multiple PR2s in kitchen environments) to upload paths to the library. Violation checking could also be performed in the cloud, which would allow us to pick the path with the least violation among all the paths in the library, instead of testing violation on only a subset of paths. Finally, multiple PFS modules could be spread across the cloud, as well as parallelizing the computation in the PFS modules and the path repair module. We intend to investigate the applicability of various types of parallel computation and cloud computing frameworks in future work.

C. Learning to Forget

One issue we have yet to address is limiting the size of the path library by forgetting paths that are not useful. Our results show that after a certain library size (depending on the type of task), adding new paths is not helpful. To address this issue, we could remove a path from the library based on how often the path is retrieved and whether the repair algorithm outperforms PFS when using this path.

VII. Conclusion

We proposed a framework, called *Lightning*, for planning paths in high-dimensional spaces that is able to learn from experience. The Lightning framework consists of two main modules, which are run in parallel: a planning-from-scratch module, and a module that retrieves and repairs paths stored in a path library. After a path is generated for a new query, a library manager decides whether to store the path based on computation time and the generated path's similarity to the retrieved path. We have demonstrated the framework on several example tasks for the PR2 and a minimally-invasive surgery robot. We found that the retrieve-and-repair module produced paths faster than planning-from-scratch in over 90% of test cases for the PR2 and in 58% of test cases for the minimally-invasive surgery robot.

VIII. Acknowledgements

This work was partially supported by the National Science Foundation grant IIS-0904672. Thanks to Ziang Xie, Abe Bachrach, and the Berkeley Laboratory for Automation Science and Engineering for feedback and discussion.

References

[1] J. Lien and Y. Lu, "Planning motion in environments with similar obstacles," in *Proc. Robotics: Science and Systems*, 2005.

[2] S. Martin, S. Wright, and J. Sheppard, "Offline and Online Evolutionary Bi-Directional RRT Algorithms for Efficient Re-Planning in Environments with Moving Obstacles," in *Proc. IEEE Conference on Automation Science and Engineering*, 2007.

[3] M. Zucker, J. Kuffner, and M. Branicky, "Multipartite RRTs for Rapid Replanning in Dynamic Environments," in *Proc. IEEE International Conference on Robotics and Automation (ICRA)*, 2007.

[4] M. Zucker, J. Kuffner, and J. A. Bagnell, "Adaptive workspace biasing for sampling-based planners," *Proc. IEEE International Conference on Robotics and Automation (ICRA)*, May 2008.

[5] C. Atkeson and J. Morimoto, "Nonparametric representation of policies and value functions: A trajectory-based approach," in *Advances in Neural Information Processing Systems (NIPS)*, 2003.

[6] M. Stolle and C. Atkeson, "Policies based on trajectory libraries," in *Proc. IEEE International Conference on Robotics and Automation (ICRA)*, 2006.

[7] Y. Tassa, T. Erez, and W. Smart, "Receding horizon differential dynamic programming," in *Advances in Neural Information Processing Systems (NIPS)*, 2008.

[8] C. Liu and C. G. Atkeson, "Standing balance control using a trajectory library," in *Proc. IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, 2009.

[9] M. Stolle, H. Tappeiner, J. Chestnutt, and C. Atkeson, "Transfer of policies based on trajectory libraries," in *Proc. IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, 2007.

[10] C. Green and A. Kelly, "Toward optimal sampling in the space of paths," in *Proc. International Symposium of Robotics Research (ISRR)*, 2007.

[11] M. S. Branicky, R. Knepper, and J. J. Kuffner, "Path and trajectory diversity: Theory and algorithms," in *Proc. IEEE International Conference on Robotics and Automation (ICRA)*, 2008.

[12] R. Knepper and M. Mason, "Empirical sampling of path sets for local area motion planning," in *Proc. International Symposium on Experimental Robotics (ISER)*. Springer, 2008.

[13] J. Lee, J. Chai, P. Reitsma, J. Hodgins, and N. Pollard, "Interactive control of avatars animated with human motion data," in *ACM Transactions on Graphics (SIGGRAPH)*, 2002.

[14] M. Lau and J. Kuffner, "Behavior planning for character animation," in *Proc. ACM SIGGRAPH/Eurographics Symposium on Computer Animation*, 2005.

[15] K. Hauser, T. Bretl, K. Harada, and J.-c. Latombe, "Using motion primitives in probabilistic sample-based planning for humanoid robots," in *Workshop on Algorithmic Foundations of Robotics (WAFR)*, 2006.

[16] N. Jetchev and M. Toussaint, "Trajectory prediction in cluttered voxel environments," in *Proc. IEEE International Conference on Robotics and Automation (ICRA)*, 2010.

[17] A. Dragan, G. Gordon, and S. Srinivasa, "Learning from Experience in Manipulation Planning: Setting the Right Goals," in *Proc. International Symposium on Robotics Research (ISRR)*, 2011.

[18] H. Sakoe and S. Chiba, "Dynamic programming algorithm optimization for spoken word recognition," *IEEE Transactions on Acoustics, Speech, and Signal Processing*, vol. ASSP-26, no. 1, 1978.

[19] J. Kuffner, J. J. and S. M. LaValle, "RRT-connect: An efficient approach to single-query path planning," in *IEEE International Conference on Robotics and Automation (ICRA)*, 2000.

[20] D. H. Jacobson and D. Q. Mayne, *Differential Dynamic Programming*. Elsevier, 1970.

[21] N. Ratliff, M. Zucker, J. A. Bagnell, and S. Srinivasa, "CHOMP: Gradient optimization techniques for efficient motion planning," in *IEEE International Conference on Robotics and Automation (ICRA)*, 2009.